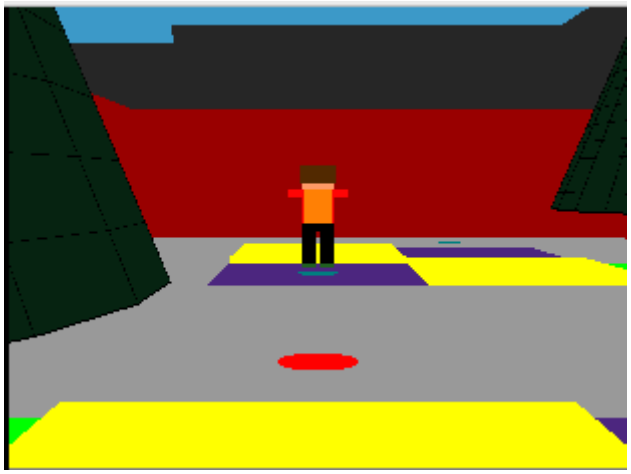
INTELIGENCIA ARTIFICIAL

E.T.S. de Ingenierías Informática y de Telecomunicación

Tutorial: Práctica 2

Agentes Reactivos/Deliberativos



DEPARTAMENTO DE CIENCIAS DE LA COMPUTACIÓN E INTELIGENCIA ARTIFICIAL

UNIVERSIDAD DE GRANADA

Curso 2023-2024

1. Introducción

El objetivo de la práctica es definir un comportamiento reactivo/deliberativo para un par de agentes cuya misión es desplazarse por un mapa hacia una casillas destino. En este tutorial se pretende introducir al estudiante en la dinámica que debería seguir para la construcción de su práctica y para ellos se resolverá casi completamente el nivel 0 de dicha práctica.

En la práctica anterior se ha desarrollado un comportamiento reactivo sobre un mundo muy parecido al que se describe y con el que trabajaremos durante la práctica 2. La mayoría de los estudiantes estarán familiarizados con estos comportamiento y con los elementos que se vuelven a repetir en esta segunda práctica. Por esta razón, indicamos a aquellos estudiantes que no tengan claro este manejo, realicen el tutorial de la práctica 1 antes de leer este. Aunque en esta segunda, los agentes reactivos no están el centro del estudio, si que son necesarios para complementar los comportamientos deliberativos que aquí se verán.

Así que para este tutorial asumiremos que se ha realizado el tutorial de la práctica anterior y nos centraremos en los pasos que se requieren para implementar un algoritmo concreto de búsqueda sobre un problema concreto a partir de la definición del concepto de estado para ese problema por un lado para conseguir un plan de acciones y por otro lado hacer notar la necesidad de definir sistemas de monitorización del plan para garantizar su correcta ejecución.

2. Estructura general del agente jugador

En la descripción de esta práctica se indica que hay dos agentes involucrados en los procesos de desplazamiento por el mapa, el agente jugador y el agente colaborador aunque en realidad los movimientos del agente colaborador son ordenados por el agente jugador y por tanto, es el comportamiento de este último el único en el que nos debemos centrar.

La estructura del software que se proporciona es semejante al utilizado en la práctica anterior, y al igual que en aquella, la forma de proceder para la realización de la tarea consiste en construir la función `think` para resolver los distintos problemas que se plantean. La función

```
Action think(Sensores sensores)
```

toma como parámetro de entrada el valor de los sensores del agente en el instante actual, y devuelve una acción. Esta forma de actuar se asemeja al comportamiento de un agente reactivo en el sentido que en cada iteración devuelve una única acción. Como en esta práctica se quieren evaluar comportamientos deliberativos necesitaremos adaptar el comportamiento de un agente deliberativo dentro de este esquema concreto. Lo que el agente hará será calcular un plan (en su versión más genérica de esta práctica), una secuencia de movimientos que permita trasladar a uno de los agente desde la casilla donde se encuentre a la casilla destino. Dicho plan se almacenará en una variable de estado. El método `think` en cada iteración de la simulación irá mandando ejecutar la siguiente acción de ese plan hasta que el agente llegue a la casilla destino.

Vamos a implementar ese comportamiento básico sobre el método `think` que viene a ejecutar la siguiente acción del plan, si el plan existe, y si no existe, se calcula el plan. Para ello vamos a definir dos variables de estado: `plan` que almacenará el plan a ejecutar y que la vamos a definir de tipo `list<Action>` en la parte privada de la clase `ComportamientoJugador` y `hayPlan` que nos dirá si ya se ha calculado un plan para llegar a la casilla objetivo que definiremos en la misma parte privada, pero en este caso su tipo asociado será `bool`. El constructor de clase la variable `hayPlan` se inicializará a `false`.

```
#ifndef COMPORTAMIENTOJUGADOR_H
#define COMPORTAMIENTOJUGADOR_H

#include "comportamientos/comportamiento.hpp"

#include <list>


class ComportamientoJugador : public Comportamiento {
  public:
    ComportamientoJugador(unsigned int size) : Comportamiento(size) {
      // Inicializar Variables de Estado
    }
    ComportamientoJugador(std::vector< std::vector< unsigned char> > mapaR) : Comportamiento(mapaR) {
      // Inicializar Variables de Estado
      hayPlan = false;
    }
    ComportamientoJugador(const ComportamientoJugador & comport) : Comportamiento(comport){}
    ~ComportamientoJugador(){}

    Action think(Sensores sensores);
    int interact(Action accion, int valor);


  private:
    // Declarar Variables de Estado
    list<Action> plan;  //Almacena el plan en ejecución
    bool hayPlan;       //Si verdad indica que se está siguiendo un plan.
};

#endif
```

A partir de las variables anteriores, podemos plantear la siguiente estructura simple para el agente jugador.

```
Action ComportamientoJugador::think(Sensores sensores){
	Action accion = actIDLE;
	if (!hayPlan){
		// Invocar al método de búsqueda
		hayPlan = true;
	}
	if (hayPlan and plan.size()>0){
		accion = plan.front();
		plan.pop_front();
	}
	if (plan.size()== 0){
		hayPlan = false;
	}
	return accion;
}
```

En el esquema anterior, bastaría con poner un procedimiento que calcule un plan para el agente, y el resto del esquema lo ejecutará hasta que el plan se termine. Indicar en el código anterior, que `Action` es el tipo de dato que codifica las acciones posibles del agente y que `actIDLE` es una constante del tipo `Action`.

Imaginemos por ejemplo que deseamos que el agente se mueva sobre el terreno como si fuese un caballo de ajedrez, es decir, que avance dos casillas en la dirección en la que se encuentra, gire a la izquierda y haga un avance más. Una función que expresa esa secuencia de acciones podría ser como la que se describe a continuación.

```
list<Action> AvanzaASaltosDeCaballo(){
      list<Action> secuencia;
      secuencia.push_back(actWALK);
      secuencia.push_back(actWALK);
      secuencia.push_back(actTURN_L);
      secuencia.push_back(actWALK);
      return secuencia;
}
```

Si en el método `think()` incluimos la llamada a `AvanzaASaltosDeCaballo()` en el lugar donde debe estar la invocación para construir el plan obtendremos algo parecido a esto:

```
1   #include "../Comportamientos_Jugador/jugador.hpp"
2   #include "motorlib/util.h"
3
4   #include <iostream>
5   #include <cmath>
6   #include <set>
7   #include <stack>
8
9   list<Action> AvanzaASaltosDeCaballo()
10  {
11      list<Action> secuencia;
12      secuencia.push_back(actWALK);
13      secuencia.push_back(actWALK);
14      secuencia.push_back(actTURN_L);
15      secuencia.push_back(actWALK);
16      return secuencia;
17  }
18
19
20  // Este es el método principal que se piden en la practica.
21  // Tiene como entrada la información de los sensores y devuelve la acción a realizar.
22  // Para ver los distintos sensores mirar fichero "comportamiento.hpp"
23  Action ComportamientoJugador::think(Sensores sensores)
24  {
25      Action accion = actIDLE;
26      if (!hayPlan)
27      {
28          cout << "Calculamos un nuevo plan\n";
29          plan = AvanzaASaltosDeCaballo ();
30          hayPlan = true;
31      }
32      if (hayPlan and plan.size() > 0)
33      {
34          accion = plan.front();
35          plan.pop_front();
36      }
37      if (plan.size() == 0)
38      {
39          cout << "Se completó el plan\n";
40          hayPlan = false;
41      }
42      return accion;
43  }
```

Si una vez que compilamos, ponemos en un terminal

```
./practica2 mapas/mapa30.map 1 0 7 7 2 10 10 4 12 5
```

y vamos pulsando al botón de *"paso"*, podremos observar como en un primer momento no hay plan, se incluye en la variable `plan` el movimiento del caballo y se aplica la primera acción de ese plan. En los siguientes 3 instantes se aplica la siguiente acción hasta que la plan se queda sin acciones. En el siguiente instante, se calcula de nuevo el plan y se ejecuta la primera acción. Este proceso se repite hasta que la simulación termine.

3. Construyendo el nivel 0

El esquema anterior en el método `think` es suficiente para llevar la ejecución de un plan cuando las condiciones del mundo son estáticas y se tiene información completa del mundo. Por tanto, el esquema anterior nos valdrá para los 4 primeros niveles de la práctica (del nivel 0 al 3) y nos podremos centrar exclusivamente en implementar el algoritmo de búsqueda que nos piden e insertarlo en el lugar del `AvanzaASaltosDeCaballo`.

En el nivel 0 debemos implementar el algoritmo de búsqueda en anchura para el problema de llevar al agente jugador a la casilla objetivo. Una descripción en pseudocódigo de este algoritmo podría ser algo como lo siguiente:

```
function BREADTH-1ST-SEARCH(problem)
begin
     current_node ← problem.INITIAL-STATE()
     frontier ← a FIFO queue
     explored ← an empty set
     frontier.insert(current_node);

     while (!frontier.empty() and !problem.Is_Solution(current_node)) do
     begin
           frontier.delete(current_node)
           explored.insert(current_node)
           for each action applicable to current_node do
           begin
                 child ← problem.apply(action, current_node)
                 if (problem.Is_Solution(child) then current_node = child
                 else if child.state() is not in explored or frontier then
                       frontier.insert(child)
           end
           if (!problem.Is_Solution(current_node) then
                 current_node ← frontier.next()
     end
     if (problem.Is_Solution(current_node)) then return SOLUTION(current_node)
     else return failure
end
```

Una descripción en lenguaje natural de ese proceso diría que se usan dos listas para recorrer el espacio de estados que se genera en la búsqueda, una lista con *alma* de COLA llamada `frontier` (que también se suele denominar `frontera` o `abiertos` en otras descripciones) que almacenará los estados pendientes de explorar y una lista con *alma* de CONJUNTO/SET llamada `explored` (o `cerrados` en otras descripciones) que almacena los estados que ya han sido explorados. La misión de la primera es mantener los estados aún no explorados en el mismo orden que se generaron en la exploración, mientras que la misión de la segunda es asegurar que un estado ya explorado no se vuelve a explorar.

Se usa una variable current_node para almacenar el estado que se está evaluando en cada momento. En el instante inicial esa variable toma el valor del estado inicial del problema.

El resto del proceso es simple de describir; mientras la lista de nodos pendientes de explorar (`frontier`) no esté vacía y el estado actual no sea una solución, entonces se hace lo siguiente: se elimina el estado actual de la `frontier` y se inserta en la lista de estados explorados (`explored`). Para cada acción aplicable sobre el estado actual, se aplica y se genera un estado descendiente (`child`). Si ese estado es ya solución, la búsqueda termina. Si no lo es y ese estado no está en ninguna de las dos listas, se inserta en `frontier`.

El procedimiento solo se saldrá del ciclo o bien porque `current_node` es solución o porque `frontier` se vació. En el primer caso se devolverá la solución asociada a ese estado. El segundo caso indica que después de explorar todos los estados que se podían generar desde el estado inicial del problema, ninguno de ellos satisfacía ser solución al problema.

3.1. Definición del concepto de estado

Ahora toca adaptar esta estrategia de búsqueda para nuestro problema y para ello debemos definir el concepto de *estado* para el nivel 0 de la práctica. La definición que se haga establecerá la complejidad/tamaño del espacio de búsqueda y por tanto, que soluciones son alcanzables y que tiempo se necesitará para alcanzarlas.

El nivel 0 indica que debemos llevar al agente jugador desde su posición inicial hasta la casilla objetivo y además el plan que se devuelva como solución no puede incluir ninguna orden de movimiento para el agente colaborador. Desde ese punto de vista parecería que el estado para este problema solo debería contener la posición del agente jugador (es decir su fila, su columna y su orientación) ya que el agente colaborador permanecerá inmóvil (sin cambiar su posición ni su orientación) en el desplazamiento del agente jugador.

El hecho que el agente colaborador no se mueva, no significa que no influya en el plan que debe construir el agente jugador ya que el primero podría estar ocupando una casilla que el segundo está considerando como transitable y por tanto que podría formar parte del camino para obtener la solución. Por esta razón, la posición del agente colaborador debe formar parte del estado del problema.

```
11  struct ubicacion{
12      int f;
13      int c;
14      Orientacion brujula;
15
16      bool operator==(const ubicacion &ub)const {
17        return (f == ub.f and c == ub.c and brujula == ub.brujula);
18      }
19  };
```

Así, definiremos estado en el problema como la ubicación del agente jugador y la ubicación del agente colaborador. En el fichero `comportamiento.hpp` se define el tipo de dato ubicación del que haremos uso para definir el concepto de estado en el problema de nivel 0.

Así, en el fichero `jugador.hpp` pasaré a definir el estado del nivel 0 como:

```
#ifndef COMPORTAMIENTOJUGADOR_H
#define COMPORTAMIENTOJUGADOR_H

#include "comportamientos/comportamiento.hpp"

#include <list>

struct stateN0{
  ubicacion jugador;
  ubicacion colaborador;
  Action ultimaOrdenColaborador;

  bool operator==(const stateN0 &x) const{
    if (jugador == x.jugador and
        colaborador.f == x.colaborador.f and colaborador.c == x.colaborador.c){
          return true;
        }
      else {
        return false;
      }
  }
};


class ComportamientoJugador : public Comportamiento {
  public:
    ComportamientoJugador(unsigned int size) : Comportamiento(size) {
      // Inicializar Variables de Estado
    }
    ComportamientoJugador(std::vector< std::vector< unsigned char> > mapaR) : Comportamien
      // Inicializar Variables de Estado
```

Sobre el tipo de dato `stateN0` defino el operador de igualdad: dos estados son iguales si la ubicación del jugador en los dos estados en la misma y la posición (sin tener en cuenta la orientación) del agente colaborador es la misma.

Se puede observar que se incluye un campo adicional `UltimaOrdenColaborador` que almacenará cual fue la última orden que el jugador le dio al colaborador. Este campo no es relevante en el nivel 0, pero lo será y mucho en los niveles que implican el movimiento del agente colaborador.

3.2. Primera aproximación a la búsqueda en anchura

Fijado el concepto de estado, pasamos a hacer una primera implementación del algoritmo de búsqueda en anchura intentando transcribir el pseudocódigo que se mostró anteriormente. El resultado sería el siguiente:

```
182  /** primera aproximación a la implementación de la búsqueda en anchura */
183  bool AnchuraSoloJugador(const stateN0 &inicio, const ubicacion &final,
184                          const vector<vector<unsigned char>> &mapa)
185  {
186      stateN0 current_state = inicio;
187      list<stateN0> frontier;
188      list<stateN0> explored;
189      frontier.push_back(current_state);
190      bool SolutionFound = (current_state.jugador.f == final.f and
191                            current_state.jugador.c == final.c);
192
193  >   while (!frontier.empty() and !SolutionFound){…
236      return SolutionFound;
237  }
```

Describiremos poco a poco esta implementación empezando por los argumentos que se han considerado. Se puede observar que tiene 3 argumentos de entrada, el estado inicial llamado `inicio` que es de tipo `stateN0`, la casilla destino a donde debe llegar el agente jugador llamada `final` y que es de tipo `ubicación` y por último el mapa sobre el que se realizará la búsqueda que se llama `mapa` y cuyo tipo es matriz de `unsigned char`. La salida de la función es un valor lógico indicando si ha sido posible encontrar el plan.

Vamos a descomponer en la descripción del algoritmo en dos partes, por un lado veremos la declaración de los datos que se usan junto con la estructura general del proceso de búsqueda, mientras que por el otro lado, veremos el cuerpo del ciclo que se dedica a la generación de los nuevos estados. Empezamos viendo la estructura general

Vemos que los datos que se usan son los siguientes:

- `current_state`: almacena el estado actual y que inicialmente toma el valor del parámetro inicio.
- `frontier`: una lista que mantiene los estados pendientes de explorar, inicialmente vacía.
- `explored`: una lista que mantiene los estados ya explorados, inicialmente vacía.
- `SolutionFound`: una variable lógica que determina si ya se ha encontrado un estado que satisface la condición de ser solución. Esta variable se inicializa mirando si `current_state satisface` la condición de ser solución, es decir, que el agente jugador esté en la fila y en la columna de la casilla destino.

Además de la declaración y antes de entrar en el ciclo principal del proceso de búsqueda (que va desde la línea 11 a la línea 40) lo único que se hace es meter el estado actual en la lista de nodos pendientes de explorar que es equivalente a meter en dicha lista el estado inicial.

Del ciclo que se plantea en este proceso se sale por dos razones: `SolutionFound` es cierto, lo que indica que se ha encontrado solución o la lista de estados pendientes de explorar está vacía, en cuyo caso, se ha explorado todo el espacio alcanzable desde el estado inicial y no ha sido posible

encontrar solución. Por tanto, el valor de `SolutionFound` determina si fue posible encontrar una solución y es el valor que se devuelve.

Ahora pasamos a describir el cuerpo del ciclo del proceso de búsqueda y en él podemos distinguir 3 partes: (1) eliminar el estado actual de la lista de pendientes de explorar y meterlo en la lista de explorados [líneas 194 y 195], (2) generar todos los descendientes del estado actual [desde la línea 197 hasta la línea 230] y (3) tomar el siguiente valor de estado actual de la lista de estados pendientes de explorar [líneas 232 y 233].

```
193     while (!frontier.empty() and !SolutionFound){
194         frontier.pop_front();
195         explored.push_back(current_state);
196 
197         // Generar hijo actWALK
198         stateN0 child_walk = apply(actWALK, current_state, mapa);
199         if (child_walk.jugador.f == final.f and child_walk.jugador.c == final.c){
200             current_state = child_walk;
201             SolutionFound = true;
202         }
203         else if (!Find(child_walk, frontier) and !Find(child_walk, explored)){
204             frontier.push_back(child_walk);
205         }
206 
207         if (!SolutionFound){
208             // Generar hijo actRUN
209             stateN0 child_run = apply(actRUN, current_state, mapa);
210             if (child_run.jugador.f == final.f and child_run.jugador.c == final.c){
211                 current_state = child_run;
212                 SolutionFound = true;
213             }
214             else if (!Find(child_run, frontier) and !Find(child_run, explored)){
215                 frontier.push_back(child_run);
216             }
217         }
218 
219         if (!SolutionFound){
220             // Generar hijo actTURN_L
221             stateN0 child_turnl = apply(actTURN_L, current_state, mapa);
222             if (!Find(child_turnl, frontier) and !Find(child_turnl, explored)){
223                 frontier.push_back(child_turnl);
224             }
225             // Generar hijo actTURN_SR
226             stateN0 child_turnsr = apply(actTURN_SR, current_state, mapa);
227             if (!Find(child_turnsr, frontier) and !Find(child_turnsr, explored)){
228                 frontier.push_back(child_turnsr);
229             }
230         }
231 
232         if (!SolutionFound and !frontier.empty())
233             current_state = frontier.front();
234     }
```

(1) En la parte de actualización de las listas se usan las operaciones primitivas para eliminar el primer elemento de una lista en el caso de `frontier` y para añadir al final de una lista en el caso de `explored`.

(2) Se puede observar que la parte destinada a la generación de los estados descendientes del estado actual es lo que ocupa la mayor parte del código. Debido a que en este nivel solo se

puede mover el agente jugador, solo 4 acciones (las que implican que dicho agente se pueda desplazar) deben ser consideradas (`actWALK`, `actRUN`, `actTURN_L` y `actTURN_SR`). El proceso que se hace para cada una de ellas es similar y para unificar la descripción nos centraremos en como se genera el descendiente de `actTURN_L` [líneas desde la 220 a la 224].

Primero se crea una nueva variable de tipo `stateN0` que almacenará el resultado de aplicar la acción correspondiente al estado actual y que en este caso se llama `child_turnl`. El cálculo del nuevo estado se hace a través de una función llamada `apply` que describiremos más tarde, pero que aquí asumiremos que nos devuelve el estado resultante de aplicar la acción.

Después se comprueba si el nuevo estado generado ya estaba en alguna de las dos listas (`frontier` o `explored`). Si no está en ninguna, entonces se añade como un nuevo estado en la lista de estados pendientes de explorar. En esta implementación incluimos una función auxiliar llamada `Find` destinada a comprobar si un elemento está o no en la lista. Más adelante mostraremos la implementación de esta función.

El generar el siguiente estado para la acción y comprobar si ya se había generado antes es común a todas las acciones que se pueden aplicar al estado actual aunque podemos observar que la acción `actWALK` tiene algo más. Por la naturaleza de este problema, las acciones `actWALK` y `actRUN` son las únicas que permite al agente cambiar de casilla, y solo cambiando de casilla es posible llegar al destino. Por esta razón, y porque en la búsqueda en anchura, cuando se genera el primer estado que satisface las condiciones de ser solución es la solución al problema y el proceso de búsqueda debe terminar, se incluye dicha verificación en ambas acciones, en concreto para actWALK son [líneas de la 199 a la 202].

Por último, indicar que en la línea 207 se incluye una condición para que no se generen los estados descendientes de los giros si ya se generó un estado que es solución.

(3) En esta última parte del cuerpo del ciclo se toma el nuevo valor para `current_state`. Solo se toma un nuevo valor, si en la generación de los descendientes no se ha encontrado un estado que satisface las condiciones de ser solución y obviamente, si la lista de nodos pendientes de explorar (`frontier`) no está vacía.

Los procesos que permiten obtener el siguiente estado a partir del estado actual se encuentra encapsulado dentro de la función `apply`. Existirían muchas posibilidades de implementación de esta función.

3.2.1. Generando los siguientes estados (`apply`)

En esta subsección se describe una posible implementación de la función `apply` que genera a partir del estado actual el estado resultante de aplicar una acción. La implementación hace uso de dos funciones auxiliares. La primera de ellas es la función `CasillaTransitable`.

```
18
19  /** Devuelve si una ubicación en el mapa es transitable para el agente*/
20  bool CasillaTransitable(const ubicacion &x, const vector<vector<unsigned char> > &mapa){
21      return (mapa[x.f][x.c] != 'P' and mapa[x.f][x.c] != 'M' );
22  }
```

Esta función toma dos argumentos de entrada, una ubicación `x` y una matriz bidimensional `mapa` y devuelve si la casilla determinada por la ubicación `x` es transitable en el `mapa`. La casilla se considera que es transitable si no es un precipicio (`'P'`) ni un muro (`'M'`).

La otra función de la que hace uso `apply` es `NextCasilla`. Esta función toma como dato de entrada una ubicación y devuelve la ubicación de la casilla que estaría delante del agente en el sentido de su avance.

```
24  /** Devuelve la ubicación siguiente según el avance del agente*/
25  ubicacion NextCasilla(const ubicacion &pos){
26      ubicacion next = pos;
27      switch (pos.brujula)
28      {
29      case norte:
30          next.f = pos.f - 1;
31          break;
32      case noreste:
33          next.f = pos.f - 1;
34          next.c = pos.c + 1;
35          break;
36      case este:
37          next.c = pos.c + 1;
38          break;
39      case sureste:
40          next.f = pos.f + 1;
41          next.c = pos.c + 1;
42          break;
43      case sur:
44          next.f = pos.f + 1;
45          break;
46      case suroeste:
47          next.f = pos.f + 1;
48          next.c = pos.c - 1;
49          break;
50      case oeste:
51          next.c = pos.c - 1;
52          break;
53      case noroeste:
54          next.f = pos.f - 1;
55          next.c = pos.c - 1;
56          break;
57      }
58
59      return next;
60  }
```

A partir de las dos funciones anteriores, planteamos `apply` como una función con tres argumentos de entrada: la acción que se quiere realizar, el estado actual del agente y el mapa del juego y devuelve como cambia el estado actual tras realizar esa acción sobre el mapa elegido.

```
63  /** Devuelve el estado que se genera si el agente puede avanzar.
64   * Si no puede avanzar, devuelve como salida el mismo estado de entrada
65  */
66  stateN0 apply(const Action &a, const stateN0 &st, const vector<vector<unsigned char> > mapa){
67      stateN0 st_result = st;
68      ubicacion sig_ubicacion, sig_ubicacion2;
69      switch (a)
70      {
71      case actWALK: //si prox casilla es transitable y no está ocupada por el colaborador
72          sig_ubicacion = NextCasilla(st.jugador);
73          if (CasillaTransitable(sig_ubicacion, mapa) and
74              !(sig_ubicacion.f == st.colaborador.f and sig_ubicacion.c == st.colaborador.c)){
75                  st_result.jugador = sig_ubicacion;
76              }
77          break;
78
79      case actRUN: //si prox 2 casillas son transitables y no está ocupada por el colaborador
80          sig_ubicacion = NextCasilla(st.jugador);
81          if (CasillaTransitable(sig_ubicacion, mapa) and
82              !(sig_ubicacion.f == st.colaborador.f and sig_ubicacion.c == st.colaborador.c)){
83                  sig_ubicacion2 = NextCasilla(sig_ubicacion);
84                  if (CasillaTransitable(sig_ubicacion2, mapa) and
85                      !(sig_ubicacion2.f == st.colaborador.f and sig_ubicacion2.c == st.colaborador.c)){
86                          st_result.jugador = sig_ubicacion2;
87                  }
88              }
89          break;
90
91      case actTURN_L:
92          st_result.jugador.brujula = static_cast<Orientacion>((st_result.jugador.brujula+6)%8);
93          break;
94
95      case actTURN_SR:
96          st_result.jugador.brujula = static_cast<Orientacion>((st_result.jugador.brujula+1)%8);
97          break;
98      }
99      return st_result;
100 }
```

La descripción de esta función es muy simple. En función de cada posible acción de las que se puede realizar (recordamos que para el nivel 0 solo 3 acciones permiten el movimiento del agente jugador) se cálcula el estado resultado de aplicar dicha acción sobre el estado `st` almacenándose el resultado final en la variable `st_result` que se inicializa con el valor de `st` al principio de la función.

En el caso de los giros, el cálculo del estado generado es simple ya que la acción no tiene precondiciones y siempre se obtiene un estado "válido". Podemos ver [en la línea 92] que consiste en aplicar una fórmula para cambiar la orientación del estado de partida sabiendo que implica dos pasos de 45 grados a la izquierda. De forma semejante se hace con el giro de 45 grados a la derecha.

El caso de la acción de avanzar si que puede implicar la generación de estados "inválidos". Un estado será inválido si se intenta avanzar a una casilla no transitable o la casilla que en este momento está ocupada por el agente colaborador. Para evitar la generación de estos estados, se fija la condición de la línea 73. Esto implica que cuando `actWALK` genera un estado inválido, se devuelve como siguiente estado el mismo estado de partida. Lo mismo ocurre con `actRUN`, pero en este caso, deben ser transitables tanto la casilla que tiene delante como la que está delante del agente pero 2 casillas más allá, por eso se realiza una doble verificación aplicando 2 veces `NextCasilla` y viendo que las dos casillas son transitables [líneas 80 y 83].

3.2.2. Función para buscar en una lista (`Find`)

```
103
104  /**Encuentra si el elemento item está en la lista */
105  bool Find(const stateN0 &item, const list<stateN0> &lista){
106      auto it = lista.begin();
107      while (it != lista.end() and !((*it) == item))
108          it++;
109
110      return (!(it == lista.end()));
111  }
```

Para que quede definitivamente terminada la implementación de la primera implementación de la búsqueda en anchura queda por mostrar la codificación de la función `Find`. Esta función tiene como argumentos de entrada un estado concreto `item` y una lista de estados y devuelve si está o no en `lista`.

Se puede observar que es una implementación de búsqueda secuencial sobre una lista y que tiene un orden de complejidad O(n), siendo n el tamaño de la lista.

3.2.3. Probando la primera implementación de la búsqueda en anchura

Una vez implementado el algoritmo de búsqueda podemos pasar a probar su funcionamiento ante algún problema de búsqueda. Para eso necesitamos invocar al algoritmo en el método `think`.

```
485  // Este es el método principal que se piden en la practica.
486  // Tiene como entrada la información de los sensores y devuelve la acción a realizar.
487  // Para ver los distintos sensores mirar fichero "comportamiento.hpp"
488  Action ComportamientoJugador::think(Sensores sensores)
489  {
490      Action accion = actIDLE;
491      if (!hayPlan)
492      {
493          cout << "Calculamos un nuevo plan\n";
494          c_state.jugador.f = sensores.posF;
495          c_state.jugador.c = sensores.posC;
496          c_state.jugador.brujula = sensores.sentido;
497          c_state.colaborador.f = sensores.CLBposF;
498          c_state.colaborador.c = sensores.CLBposC;
499          c_state.colaborador.brujula = sensores.CLBsentido;
500          goal.f = sensores.destinoF;
501          goal.c = sensores.destinoC;
502
503          hayPlan = AnchuraSoloJugador(c_state, goal, mapaResultado);
504          if (hayPlan) cout << "Se encontró un plan\n";
505      }
506      if (hayPlan and plan.size() > 0)
507      {
508          accion = plan.front();
509          plan.pop_front();
510      }
511      if (plan.size() == 0)
512      {
513          cout << "Se completó el plan\n";
514          hayPlan = false;
515      }
516      return accion;
517  }
```

Esta sería la nueva configuración del método `think`, donde se modifica todo el bloque relativo al cálculo del nuevo plan [desde las líneas 494 a la 505]. Previamente a invocar al algoritmo de búsqueda [líneas de la 494 a la 501] se rellenan las variables `c_state` de tipo `stateN0` y `goal` de tipo `ubicacion` que almacenan el estado actual (ubicación del agente jugador y del agente colaborador) y la casilla destino respectivamente.

Las variables `c_state` y `goal` son variables de estado que se declaran en la parte privada de la clase `ComportamientoJugador` en el fichero `jugador.hpp`.

Por último, en línea 503 se invoca al algoritmo de búsqueda pasando como argumentos de entrada `c_state`, `goal` y `mapaResultado` (la matriz bidimensional donde se encuentra codificado el mapa) y se devuelve un valor lógico. En línea 504 se saca el mensaje "`Se encontró un plan`" si el resultado de invocar al método de búsqueda devuelve `true`.

Para probar el funcionamiento, compilamos y ejecutamos con el siguiente comando

```
./practica2 mapas/mapa30.map 1 0 5 5 2 10 10 4 12 5
```

Observamos que no aparece que nada haya cambiado incluso si continuamos pulsando el botón de "paso" nada parece cambiar. Solo podemos ver en la consola del terminar mensajes indicando que se calcula un nuevo plan y que se completó dicho plan.

La razón es que el algoritmo solo verifica que la casilla objetivo es alcanzable desde la casilla del agente jugador, pero no construye y devuelve la secuencia de acciones que llevan a que se produzca ese desplazamiento. Por tanto, necesitamos cambiar el algoritmo de búsqueda para que devuelva esa secuencia de acciones.

3.3. Segunda aproximación a la búsqueda en anchura

La primera implementación de la búsqueda en anchura ha dado como resultado un proceso que determina si es posible o no encontrar una secuencia de acciones para llevar al agente jugador desde su casilla a la casilla destino pero no devuelve el plan concreto que consigue eso.

En esta segunda aproximación incluiremos los cambios necesarios en la primera versión del algoritmo de búsqueda para devolver el plan de acciones cuando sea posible encontrar ese plan. Antes vamos a definir una función que nos permitirá ver gráficamente el plan que se ha obtenido sobre el simulador. Posteriormente incluiremos las modificaciones necesarias en el algoritmo de búsqueda para obtener el plan.

3.3.1. Una función para representar visualmente el plan en el simulador

Vamos a definir aquí una función que nos permitirán ver en la parte gráfica del simulador los planes que se vayan construyendo por parte del agente jugador. Para su implementación hemos hecho uso de una función auxiliar llamada `AnulaMatriz`.

```
114  /** pone a cero todos los elementos de una matriz */
115  void AnulaMatriz(vector<vector<unsigned char> > &matriz){
116      for (int i = 0; i < matriz.size(); i++)
117          for (int j = 0; j < matriz[0].size(); j++)
118              matriz[i][j] = 0;
119  }
120
```

La función `AnulaMatriz` toma como entrada una matriz bidimensional de `unsigned char` que modifica para poner todas sus componentes a cero.

La función de `VisualizaPlan` tiene dos argumentos, un dato de tipo `stateN0` que nos indica la ubicación de los dos agentes (jugador y colaborador) en el momento que fue elaborado el plan y un plan a realizar. El resultado es que la función modifica una matriz interna del simulador llamada `mapaConPlan`, del tamaño del mapa del juego donde el valor 0 indica que no será una casilla transitada por el plan, el valor 1 que será una casilla transitada por el agente jugador, el valor 2 que será transitada por el agente colaborador y el valor 3 si es una casilla intermedia en una acción de `actRUN` del jugador.

La función recorre el plan sobre `mapaConPlan` para determinar las casillas que serán usadas por los dos agentes y las marca en esa matriz.

Como se puede observar, el campo `ultimaOrdenColaborador` es clave en el estado y fundamental para poder establecer como avanzará el plan. Esta variable conserva la última acción que el jugador le ordenó al colaborador, y si en un instante el jugador no le indica una nueva acción a realizar, el colaborador aplicará la última que se le indicó. Así, si el agente jugador indicó un `act_CLB_WALK` primero y luego un `actWALK`, en el segundo instante el agente colaborador aplicará un nuevo avance a la vez que el jugador también se desplaza.

```
122  // Permite pintar sobre el mapa del simulador el plan partiendo desde el estado st
123  void ComportamientoJugador::VisualizarPlan(const stateN0 &st, const list<Action> &plan)
124  {
125      AnularMatriz(mapaConPlan);
126      stateN0 cst = st;
127
128      auto it = plan.begin();
129      while (it != plan.end())
130      {
131          if ((*it != act_CLB_WALK) and (*it != act_CLB_TURN_SR) and (*it != act_CLB_STOP))
132          {
133              switch (cst.ultimaOrdenColaborador)
134              {
135              case act_CLB_WALK:
136                  cst.colaborador = NextCasilla(cst.colaborador);
137                  mapaConPlan[cst.colaborador.f][cst.colaborador.c] = 2;
138                  break;
139              case act_CLB_TURN_SR:
140                  cst.colaborador.brujula = (Orientacion)((cst.colaborador.brujula + 1) % 8);
141                  break;
142              }
143          }
144
145          switch (*it)
146          {
147          case actRUN:
148              cst.jugador = NextCasilla(cst.jugador);
149              mapaConPlan[cst.jugador.f][cst.jugador.c] = 3;
150              cst.jugador = NextCasilla(cst.jugador);
151              mapaConPlan[cst.jugador.f][cst.jugador.c] = 1;
152              break;
153          case actWALK:
154              cst.jugador = NextCasilla(cst.jugador);
155              mapaConPlan[cst.jugador.f][cst.jugador.c] = 1;
156              break;
157          case actTURN_SR:
158              cst.jugador.brujula = (Orientacion)((cst.jugador.brujula + 1) % 8);
159              break;
160          case actTURN_L:
161              cst.jugador.brujula = (Orientacion)((cst.jugador.brujula + 6) % 8);
162              break;
163          case act_CLB_WALK:
164              cst.colaborador = NextCasilla(cst.colaborador);
165              cst.ultimaOrdenColaborador = act_CLB_WALK;
166              mapaConPlan[cst.colaborador.f][cst.colaborador.c] = 2;
167              break;
168          case act_CLB_TURN_SR:
169              cst.colaborador.brujula = (Orientacion)((cst.colaborador.brujula + 1) % 8);
170              cst.ultimaOrdenColaborador = act_CLB_TURN_SR;
171              break;
172          case act_CLB_STOP:
173              cst.ultimaOrdenColaborador = act_CLB_STOP;
174              break;
175          }
176          it++;
177      }
178  }
```

3.3.2. De estado a nodo

Normalmente, en la implementación de los algoritmos de búsqueda es necesario incluir más información que la del estado para poder realizar las tareas que indica dicho algoritmo. A esa implementación de un estado con más información se le conoce con el nombre de nodo y suele ser muy habitual, cuando se describe un algoritmo de búsqueda, usar el concepto de nodo.

En nuestro caso, usar solo la información del estado para el problema de búsqueda es lo que no nos permite almacenar la secuencia de acciones. Por esa razón, vamos a definir un nodo, que obviamente incluirá al estado, pero además información para construir el plan. Existen versiones que enlazan los nodos con apuntadores a los nodos padres que los generaron y reconstruyen el plan recorriendo esos enlaces desde el nodo que contiene un estado solución hasta el nodo que contienen el estado inicial. En lugar de esa opción, nosotros escogeremos una versión más simple que consiste en almacenar en cada nodo la secuencia de acciones que se han aplicado para llegar a ese nodo. De esta manera, una vez que se encuentra el nodo solución, el plan es devolver justo esa secuencia de acciones.

```
241
242  /**Definición del tipo nodo del nivel 0*/
243  struct nodeN0{
244      stateN0 st;
245      list<Action> secuencia;
246
247      bool operator==(const nodeN0 &n) const {
248          return (st == n.st);
249      }
250
251      bool operator<(const nodeN0 &b)  const {
252          if (st.jugador.f < b.st.jugador.f)
253              return true;
254          else if (st.jugador.f == b.st.jugador.f and st.jugador.c < b.st.jugador.c)
255              return true;
256          else if (st.jugador.f == b.st.jugador.f and st.jugador.c == b.st.jugador.c and st.jugador.brujula < b.st.jugador.brujula)
257              return true;
258          else
259              return false;
260      }
261  };
262
```

Podemos observar que hemos definido `nodeN0` como un `struct` con 2 campos, `st` que codifica el estado y secuencia que almacenará la secuencia de acción que han sido necesarias desde el estado inicial para llegar al estado codificado en `st`.

Además definimos el operador de igualdad, indicando que dos nodos son iguales si los estados que codifican son iguales. También se ha definido el operador de menor estricto. La razón se verá más adelante. Aquí de momento solo indicar que se considera que dados 2 nodos, se considera menor el que tiene menos valor de fila en el agente jugador, a igualdad en el valor de fila el que tenga menor valor de columna en el agente jugador y a igualdad en filas y columnas en la ubicación del agente jugador, el que la codificación ordinal de su orientación sea menor también en el agente jugador. Como se puede observar, la ubicación del agente colaborador se considera irrelevante para determinar la diferencia entre dos estados. La razón es que en el nivel 0, la ubicación del agente colaborador solo se incluye para evitar que el agente jugador acceda a la casilla que ocupa el colaborador, pero el agente colaborador en sí no cambiará su posición o no al menos por una acción realizada por el agente jugador.

Dada la definición anterior de nodo, la segunda versión que se propone del algoritmo de búsqueda consiste en modificar la primera versión para incluir en el lugar de `stateN0`, `nodeN0` y con ello poder tener el plan. Obviamente, otros cambios son necesarios. Las listas que inicialmente eran de `stateN0` ahora pasan a ser de `nodeN0`, y por tanto, para buscar estados dentro de las listas de nodos tendremos que sobrecargar la función `Find` que se encarga de encontrar si estado está o no en la `lista` de nodos pendientes de explorar o nodos explorados.

```
263
264  /**Sobrecarga de la función Find para buscar en lista de nodeN0*/
265  bool Find(const stateN0 &item, const list<nodeN0> &lista){
266      auto it = lista.begin();
267      while (it != lista.end() and !(it->st == item))
268          it++;
269
270      return (!(it == lista.end()));
271  }
272
```

Se puedo observar que la función es una reformulación de la inicial sabiendo ahora que la lista es de nodos y que el estado se codifica en el campo `st` de dicho nodo. Al igual que la inicial, el orden de complejidad es de O(n) siendo n el tamaño de la lista.

Se incluye también la función `PintaPlan` para mostrar en formato de texto la secuencia de acciones que constituye el plan obtenido. No se incluye ninguna descripción adicional sobre esta función ya que su funcionamiento se entiende de forma fácil.

```
275
276
277  void PintaPlan(const list<Action> &plan)
278  {
279      auto it = plan.begin();
280      while (it != plan.end())
281      {
282          if (*it == actWALK){
283              cout << "W ";
284          }
285          else if (*it == actRUN){
286              cout << "R ";
287          }
288          else if (*it == actTURN_SR){
289              cout << "r ";
290          }
291          else if (*it == actTURN_L){
292              cout << "L ";
293          }
294          else if (*it == act_CLB_WALK){
295              cout << "cW ";
296          }
297          else if (*it == act_CLB_TURN_SR){
298              cout << "cr ";
299          }
300          else if (*it == act_CLB_STOP){
301              cout << "cS ";
302          }
303          else if (*it == actIDLE){
304              cout << "I ";
305          }
306          else{
307              cout << "-_ ";
308          }
309          it++;
310      }
311      cout << " (" << plan.size() << " acciones)\n";
312  }
```

3.3.3. Segunda versión del algoritmo de búsqueda en anchura

Ya tenemos todos los elementos para plantear la segunda versión del algoritmo de búsqueda en anchura y en este caso si que podremos devolver la secuencia de acciones si el plan se encuentra.

```
325  /**Version segunda de la búsqueda en anchura: ahora sí devuelve un plan*/
326  list <Action> AnchuraSoloJugador_V2(const stateN0 &inicio, const ubicacion &final,
327                                      const vector<vector<unsigned char >> &mapa){
328      nodeN0 current_node;
329      list<nodeN0> frontier;
330      list<nodeN0> explored;
331      list<Action> plan;
332      current_node.st = inicio;
333      bool SolutionFound = (current_node.st.jugador.f == final.f and
334                            current_node.st.jugador.c == final.c);
335      frontier.push_back(current_node);
336
337  >   while (!frontier.empty() and !SolutionFound){…
387      if (SolutionFound){
388          plan = current_node.secuencia;
389          cout << "Encontrado un plan: ";
390          PintaPlan(current_node.secuencia);
391      }
392
393      return plan;
394  }
```

La primera novedad que se puede ver es que ahora la función devuelve una lista de acciones en lugar de una valor lógico, es decir, que ahora el resultado de ejecutar la función será la secuencia de acciones si es posible encontrar un plan.

En relación a la declaración de los datos que se usarán en el algoritmo, vemos que lo que era el estado actual (`current_state`) es sustituido por el nodo actual (`current_node`) que inicializa su campo `st` con el estado inicial (inicio, línea 332). Las listas de estados se han convertido en listas de nodos tanto `frontier`, para los estados pendientes de ser explorados, como `explored`, con los estados ya explorados). Como en la versión primera, el nodo inicial (antes `current_state`) se inserta en `frontier` y se inicializaba la variable `SolutionFound` mirando si ya el estado inicial es solución.

Se puede observar que el ciclo principal se sigue manteniendo con las mismas condiciones y más adelante describiremos los cambios que básicamente están relacionados con la adaptación del paso de estado a nodo.

En la parte final del algoritmo ahora sí que podemos ver un cambio. Cuando se encuentra solución se devuelve el plan contenido en el campo `secuencia` del nodo encontrado como solución.

```
337     while (!frontier.empty() and !SolutionFound){
338         frontier.pop_front();
339         explored.push_back(current_node);
340 
341         // Generar hijo actWALK
342         nodeN0 child_walk = current_node;
343         child_walk.st = apply(actWALK, current_node.st, mapa);
344         child_walk.secuencia.push_back(actWALK);
345         if (child_walk.st.jugador.f == final.f and child_walk.st.jugador.c == final.c){
346             current_node = child_walk;
347             SolutionFound = true;
348         }
349         else if (!Find(child_walk.st, frontier) and !Find(child_walk.st, explored)){
350             frontier.push_back(child_walk);
351         }
352 
353         if (!SolutionFound){
354             // Generar hijo actRUN
355             nodeN0 child_run = current_node;
356             child_run.st = apply(actRUN, current_node.st, mapa);
357             child_run.secuencia.push_back(actRUN);
358             if (child_run.st.jugador.f == final.f and child_run.st.jugador.c == final.c){
359                 current_node = child_run;
360                 SolutionFound = true;
361             }
362             else if (!Find(child_run.st, frontier) and !Find(child_run.st, explored)){
363                 frontier.push_back(child_run);
364             }
365         }
366 
367         if (!SolutionFound){
368             // Generar hijo actTURN_L
369             nodeN0 child_turnl = current_node;
370             child_turnl.st = apply(actTURN_L, current_node.st, mapa);
371             child_turnl.secuencia.push_back(actTURN_L);
372             if (!Find(child_turnl.st, frontier) and !Find(child_turnl.st, explored)){
373                 frontier.push_back(child_turnl);
374             }
375             // Generar hijo actTURN_SR
376             nodeN0 child_turnsr = current_node;
377             child_turnsr.st = apply(actTURN_SR, current_node.st, mapa);
378             child_turnsr.secuencia.push_back(actTURN_SR);
379             if (!Find(child_turnsr.st, frontier) and !Find(child_turnsr.st, explored)){
380                 frontier.push_back(child_turnsr);
381             }
382         }
383 
384         if (!SolutionFound and !frontier.empty())
385             current_node = frontier.front();
386     }
```

Mirando ahora el contenido del cuerpo del ciclo principal del proceso de búsqueda podemos observar que la estructura se mantiene con las 3 partes: (1) la actualización de las listas de `frontier` y `explored` que se mantienen igual [líneas 338 y 339], (2) el cálculo de aplicar todas las acciones al estado actual para generar todos los estados descendientes que básicamente permanece igual [líneas de la 341 a la 382] y (3) y sacar el siguiente estado actual de la lista `frontier` si no se ha encontrado solución y la lista no está vacía [líneas 384 y 385].

En realidad solo se ha modificado de la parte del cálculo de la siguiente acción la adaptación a que ahora `current_node` es de tipo `nodeN0` y no de tipo `stateN0`. Por eso en línea 369 `child_turnl` es de tipo `nodeN0` y se le asigna el actual nodo, en línea 370 la generación del nuevo estado sigue haciendo uso de la función `apply`, pero se le pasa como argumento solo el estado (el campo `st` de

nodo) y se asigna también al campo `st` y en línea 372 se hace uso de la función `Find` que sobrecargamos para buscar en listas de nodos.

En línea 371 si aparece una verdadera novedad que es la actualización del campo `secuencia` del nodo. En ella se añade a la secuencia que venía del padre esta nueva acción que acabamos de aplicar. Esto se hace cuando ese estado no había sido generado previamente.

Hemos explicado las modificaciones en el caso de aplicar la acción `actTURN_L`, para `actWALK`, `actRUN` y `actTURN_SR` las modificaciones son semejantes.

3.3.4. Probando la segunda implementación de la búsqueda en anchura

Ya tenemos codificado completamente el algoritmo de búsqueda en anchura y en este caso nos devolverá la secuencia de acciones si encuentra solución. Ahora pasaremos a probarlo: en el método `think` sustituimos la anterior llamada de

```
        hayPlan = AnchuraSoloJugador(c_state, goal, mapaResultado);
        if (hayPlan) cout << "Se encontró un plan\n";
```

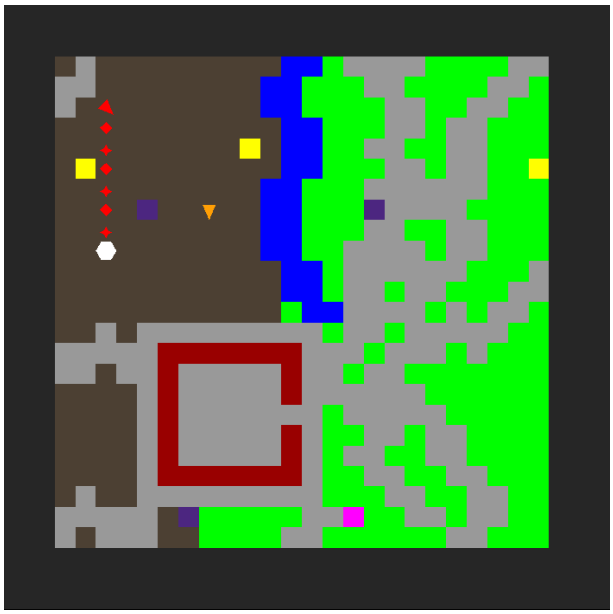
por

```
        plan = AnchuraSoloJugador_V2(c_state, goal, mapaResultado);
        VisualizaPlan(c_state,plan);
        hayPlan = true;
```

compilamos y ejecutamos

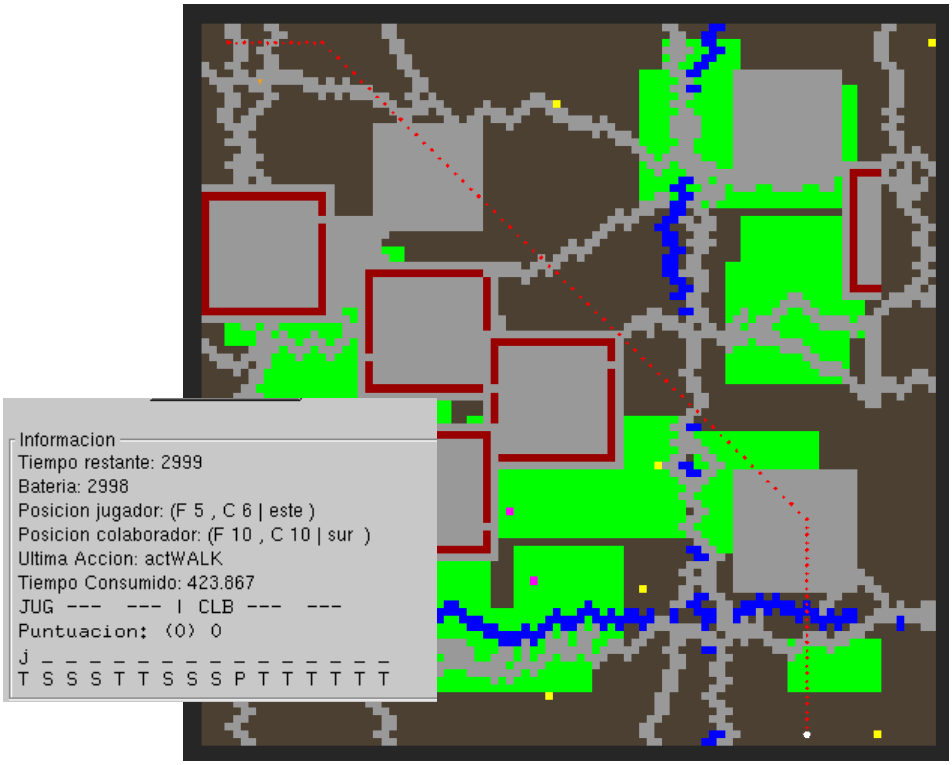
```
./practica2 mapas/mapa30.map 1 0 5 5 2 10 10 4 12 5
```

y obtenemos el siguiente resultado, un plan de longitud 6.



Repetimos la ejecución pero ahora con la siguiente llamada

```
./practica2 mapas/mapa100.map 1 0 5 5 2 10 10 4 95 80
```



En este caso, tarda unos 423 segundos en conseguir el plan (este valor de tiempo depende del ordenador donde se ha ejecutado y por tanto puede darte un valor distinto). El plan está compuesto por 54 acciones.

3.4. Tercera versión de la búsqueda en anchura: acelerando la búsqueda

En esta tercera y última versión, modificaremos ligeramente el método de búsqueda para mejorar el tiempo de respuesta del algoritmo. Las modificaciones afectarán fundamentalmente a las estructuras de datos que utilizamos para almacenar las dos listas. La lista de nodos pendientes de explorar (`frontier`) está implementada siguiendo el TDA lista. Las operaciones que requiere el algoritmo de búsqueda sobre esta lista son 4: insertar, leer el primero, sacar el primero y buscar. Las 3 primeras operaciones son muy eficientes y se podrían haber hecho usando un TDA cola (`queue`), pero en la implementación de `queue` en la `STL` de `C++` no tiene un operador para buscar y por eso razón se usó la TDA de lista usándola como si fuera una cola (el primero en entrar es el primero en salir).

La búsqueda de elementos en una lista tiene orden de complejidad O(n) como ya hemos referido varias veces a lo largo de este tutorial y no resulta un mecanismo muy eficiente si se requiere hacerlo muchas veces y si las listas son muy grandes como es el caso de este algoritmo de búsqueda. Se requiere buscar en `frontier` cada vez que se genera un nuevo estado.

En esta versión del algoritmo de búsqueda en anchura que aquí proponemos, vamos a eliminar la búsqueda de nodos en `frontier`, así que mantenemos el tipo `list` asociado a esta lista y todas las búsquedas de nodos se harán sobre la lista `explored`.

Si todas las búsquedas las haremos sobre `explored`, se deberá usar una estructura de datos eficiente para realizar esta operación en dicha lista. Vamos a optar por usar el TDA `set` para su implementación. Sobre esta estructura de datos, las operaciones de búsqueda e inserción son de $O(\log_2(n))$ que son justo las operaciones que el algoritmo de búsqueda en anchura requiere.

El tipo de dato `set` de la `STL` de `C++` requiere definir un operador de orden entre los elementos que forman parte de la lista, ya que en la base de su implementación hay un árbol binario de búsqueda. Por esa razón, cuando definimos el tipo `nodeN0`, definimos el operador *menor estricto*. Es este operador el que usará el tipo `set` para ordenar los elementos dentro de la estructura. Recordemos como estaba definido:

```
251     bool operator<(const nodeN0 &b)  const {
252         if (st.jugador.f < b.st.jugador.f)
253             return true;
254         else if (st.jugador.f == b.st.jugador.f and st.jugador.c < b.st.jugador.c)
255             return true;
256         else if (st.jugador.f == b.st.jugador.f and st.jugador.c == b.st.jugador.c and st.jugador.brujula < b.st.jugador.brujula)
257             return true;
258         else
259             return false;
260     }
```

Aunque debe tener estructura de relación de orden por los requerimientos del `set`, desde el punto de vista del papel que juega dentro del algoritmo de búsqueda en realidad se puede ver como un operador de igualdad, y con esa visión se podría leer como: *dos nodos son distintos si la fila del agente jugador de un nodo es menor que la fila del agente jugador del otro nodo. También son distintos si siendo lo anterior igual, la columna del agente jugador de un nodo es menor que la*

columna del agente jugador del otro nodo. Por último, son distintos si siendo las filas y las columnas iguales para los dos nodos, la orientación del agente jugador de un nodo es menor que la del agente jugador del otro nodo. Si no pasa lo que se ha descrito antes, los dos nodos son iguales.

Bueno, ya hemos tomado la decisión sobre los tipos de datos de las listas, `frontier` seguirá siendo una lista de `nodeN0` sobre el que no se hará ninguna búsqueda y `explored` será un `set` de `nodeN0` y solo sobre él se realizarán todas las búsquedas para encontrar los nodos repetidos.

```
398  /**Version tercera de la búsqueda en anchura*/
399  list <Action> AnchuraSoloJugador_V3(const stateN0 &inicio, const ubicacion &final,
400                                      const vector<vector<unsigned char >> &mapa){
401      nodeN0 current_node;
402      list<nodeN0> frontier;
403      set<nodeN0> explored;
404      list<Action> plan;
405      current_node.st = inicio;
406      bool SolutionFound = (current_node.st.jugador.f == final.f and
407                            current_node.st.jugador.c == final.c);
408      frontier.push_back(current_node);
409
410
411  >   while (!frontier.empty() and !SolutionFound){…
467
468      if (SolutionFound){
469          plan = current_node.secuencia;
470          cout << "Encontrado un plan: ";
471          PintaPlan(current_node.secuencia);
472      }
473
474      return plan;
475  }
```

Mirando la estructura más externa podemos observar que solo hay un cambio con respecto a la versión 2 y es en la línea 403 donde ahora `explored`, la lista de nodos ya explorados, se define de tipo `set`.

```
411      while (!frontier.empty() and !SolutionFound){
412          frontier.pop_front();
413          explored.insert(current_node);
440
441          if (!SolutionFound){
442              // Generar hijo actTURN_L
443              nodeN0 child_turnl = current_node;
444              child_turnl.st = apply(actTURN_L, current_node.st, mapa);
445              child_turnl.secuencia.push_back(actTURN_L);
446              if (explored.find(child_turnl) == explored.end()){
447                  frontier.push_back(child_turnl);
448              }
449              // Generar hijo actTURN_SR
450              nodeN0 child_turnsr = current_node;
451              child_turnsr.st = apply(actTURN_SR, current_node.st, mapa);
452              child_turnsr.secuencia.push_back(actTURN_SR);
453              if (explored.find(child_turnsr) == explored.end()){
454                  frontier.push_back(child_turnsr);
455              }
456          }
457
458          if (!SolutionFound and !frontier.empty()){
459              current_node = frontier.front();
460              while (!frontier.empty() and explored.find(current_node) != explored.end()){
461                  frontier.pop_front();
462                  if (!frontier.empty())
463                      current_node = frontier.front();
464              }
465          }
466      }
467
```

Las cosas que cambian dentro del ciclo principal del algoritmo de búsqueda en anchura [desde la línea 411 a la 467] son:

- la operación de añadir un nuevo nodo a la `explored` (antes al ser una lista se usaba el método `push_back` y ahora al ser un `set` se usa el método `insert`) en la línea 413.
- no se busca en las dos listas sino que solo se busca en la lista de `explored` y eso se expresa en las líneas 423, 436, 446 y 453.
- el proceso para elegir el siguiente nodo actual [líneas de la 458 a 464]. Para entender esto necesitamos conocer que implica no buscar nodos en `frontier`.

  Antes de esta modificación no había nodos repetidos ni en `frontier` ni en `explored`. Ahora puede suceder que varias instancias de un mismo nodo aparezcan múltiples veces en dicha lista. En principio, esto no implicaría un comportamiento distinto al algoritmo de búsqueda en anchura siempre y cuando antes de tomar un nodo como nodo actual, se verifique que no es un nodo que ya se haya explorado. Pues justo eso es lo que hace el trozo de código de las líneas anteriormente referidas, van tomando nodos de `frontier` en el mismo orden en el que fueron añadidos a la lista hasta encontrar el primero de ellos que verifica no estar ya en `explored`. Con esta modificación, lo que hacemos es permitir que la lista `frontier` ocupe potencialmente mucho más espacio a cambio de hacer menos procesos de búsqueda con la intención de reducir el tiempo necesario para encontrar una solución.

Vamos a probar el funcionamiento de esta tercera versión, y para ello sustituimos

```
hayPlan = AnchuraSoloJugador_V2(c_state, goal, mapaResultado);
```

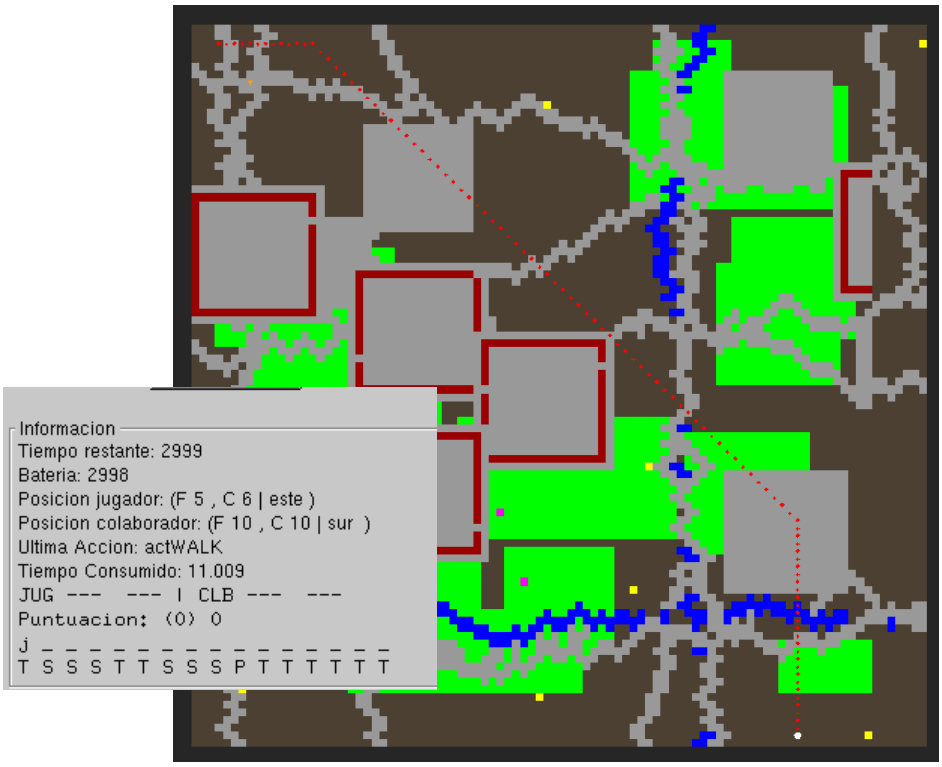
por

```
hayPlan = AnchuraSoloJugador_V3(c_state, goal, mapaResultado);
```

compilamos y ejecutamos

```
./practica2 mapas/mapa100.map 1 0 5 5 2 10 10 4 95 80
```

obtenemos



Se puede observar que el tiempo que se ha obtenido es de unos 11 segundos frente a los 423 que obtuvo la versión anterior. El plan resultante contiene 54 acciones.

4. Estructurando el método `think`

La sección anterior la hemos dedicado a describir como desarrollar el nivel 0 de la práctica 2 partiendo del software que se puede descargar desde https://github.com/ugr-ccia-IA/practica2. Pero el nivel 0 es sólo uno de los niveles que el estudiante tiene que resolver de la práctica 2 y por esa razón vamos a proponer una estructura del método `think` que permita añadir con facilidad la implementaciones del resto de niveles que quedan por completar.

Los niveles 1, 2 y 3 tienen un enunciado muy parecido al del nivel 0 y por tanto tienen una forma de resolución muy parecida, al punto, que resolverlos consiste en proponer un nuevo módulo (una función) con una estructura muy parecida a `AnchuraSoloJugador_V3`.

Sin embargo el nivel 4 si que puede ser muy diferente a los demás, ya que será necesario combinar comportamientos reactivos con deliberativos teniendo en cuenta que ahora el entorno es dinámico y que hace que pueda que los planes no se puedan terminar con éxito.

Por tanto, para enganchar las soluciones de los niveles del 1 al 3 con el programa principal, bastaría asignar en la líneas 506, 509 y 512 la llamada al módulo que resuelve el nivel 1, 2 y 3 respectivamente. La llamada sería semejante a la del nivel 0, es decir, asignar a la variable de estado plan la secuencia de acciones que se obtienen para llevar al agente correspondiente dependiendo del nivel a la ubicación destino.

```
488  Action ComportamientoJugador::think(Sensores sensores)
489  {
490      Action accion = actIDLE;
491      if (sensores.nivel != 4){
492          if (!hayPlan)
493          {
494              cout << "Calculamos un nuevo plan\n";
495              c_state.jugador.f = sensores.posF;
496              c_state.jugador.c = sensores.posC;
497              c_state.jugador.brujula = sensores.sentido;
498              c_state.colaborador.f = sensores.CLBposF;
499              c_state.colaborador.c = sensores.CLBposC;
500              c_state.colaborador.brujula = sensores.CLBsentido;
501              goal.f = sensores.destinoF;
502              goal.c = sensores.destinoC;
503              switch (sensores.nivel){
504              case 0: plan = AnchuraSoloJugador_V3 (c_state, goal, mapaResultado);
505                  break;
506              case 1: // Incluir aqui la llamada al alg. búsqueda del nivel 1
507                  cout << "Pendiente de implementar el nivel 1\n";
508                  break;
509              case 2: // Incluir aqui la llamada al alg. búsqueda del nivel 2
510                  cout << "Pendiente de implementar el nivel 2\n";
511                  break;
512              case 3: // Incluir aqui la llamada al alg. búsqueda del nivel 3
513                  cout << "Pendiente de implementar el nivel 3\n";
514                  break;
515              }
516              if (plan.size() > 0){
517                  VisualizaPlan(c_state, plan);
518                  hayPlan = true;
519              }
520          }
521          if (hayPlan and plan.size() > 0){
522              accion = plan.front();
523              plan.pop_front();
524          }
525          if (plan.size() == 0){
526              cout << "Se completó el plan\n";
527              hayPlan = false;
528          }
529      }
530      else {
531          // Incluir aqui la solución al nivel 4
532      }
533      return accion;
534  }
```

La resolución del nivel 4 se hará en el bloque del else de la línea 530.

Decir que usar esta organización del método `think` no es obligatoria, solo se da como orientación. Cualquier otra organización sensata, que respete los criterios de las buenas prácticas de programación y que lleve a conseguir las soluciones que se plantean en los distintos niveles obviamente también será válida.

5. Comentarios Finales

Este tutorial tiene como objetivo dar un pequeño empujón en el inicio del desarrollo de la práctica y lo que se propone es sólo una forma de dar respuesta (la más básica en todos los casos) a los problemas con los que os tenéis que enfrentar. Por tanto, todo lo que se propone aquí es mejorable y lo debéis mejorar.

Muchos elementos que forman parte de la práctica no se han tratado en este tutorial. Esos elementos son relevantes para mejorar la capacidad del agente e instamos a que se les preste atención.

Por último, resaltar que la práctica es individual y que la detección de copias (trozos de código iguales o muy parecidos entre estudiantes) implicará el suspenso en la asignatura.